Enabling Social Applications via Decentralized Social Data Management

Nicolas Kourtellis, Jeremy Blackburn, Cristian Borcea, and Adriana Iamnitchi

Abstract—Online social networks and user-generated content sharing applications produce an unprecedented amount of social information, which is further augmented by location or collocation data collected from mobile phones. This wealth of social information is currently fragmented across many different proprietary applications. Combined, it could provide a more accurate representation of the social world that can be leveraged to enable novel socially-aware applications. We present Prometheus, a peer-to-peer service that collects social information from multiple sources and exposes it through an interface that implements non-trivial social inferences. The social information is stored in a multigraph which is managed in a decentralized fashion on user-contributed nodes. The access to social data is controlled by user-defined policies.

The system's socially-aware design serves multiple purposes. First, it allows users to store and manage their social information via socially-trusted peers, thus improving service availability. Second, it exploits naturally-formed social groups for improved end-to-end social inference performance and reduced message overhead. Third, it reduces the opportunity of malicious peers to influence requests in the system, thus constituting a resilient solution to malicious attacks. We tested our prototype on PlanetLab under workloads from emulated applications. We also built a mobile social application to assess Prometheus' performance under real-time constraints and show that Prometheus' overhead is practical for real applications. Our experimental results show that a socially-aware distribution of social data onto Prometheus nodes leads to significant improvements in end-to-end response time, reduced communication overhead, and improved resilience to malicious attacks.

Index Terms—social data management, decentralized social graph, P2P networks, social sensors, social inferences

1 Introduction

Recent socially-aware applications leverage users' social information to provide features such as filtering restaurant recommendations based on reviews by friends (e.g., Yelp), recommending email recipients or filtering spam based on previous email activity [1], and exploiting social incentives for computer resource sharing [2], [3]. Social information is also leveraged in conjunction with location and collocation data in mobile applications such as Loopt, Foursquare and Google's Latitude. These applications collect, store, and use sensitive social information.

The state of the art is to collect and manage such information within an application, thus offering social functionalities limited only to the context of the application, as in the examples above, or to expose this information from platforms that specifically collect it, such as online social networks (OSNs). For example, Facebook (via Graph API [4]) and Google (via OpenSocial [5]) allow 3rd-party application developers and websites to access the social information of millions of users stored in their OSNs. However, hidden incentives for users to have many OSN "friends" leads to declarations of contacts with little connection in terms of trust, common interests, shared objectives, or other such manifestations of real social relationships [6]. Thus, an application that tries to provide targeted functionalities using social information exposed by current OSNs must wade through a lot of noise.

This paper presents Prometheus, a peer-to-peer (P2P) service that collects social information from multiple sources and exposes it via an API that implements social inferences which can enable a wide range of new socially-aware applications. This service collects social information from actual interactions between users within multiple environments and under multiple identities (e.g., OSNs, email, mobile phones). Thus, it maintains richer and more nuanced social information than current OSNs or social applications, which can lead to more accurate inferences of trust, interests, and context. Prometheus represents social information as a directed, weighted and labeled social multi-graph distributed on a P2P platform. Access to such aggregated social information is controlled by user-defined policies.

The choice of a P2P architecture was motivated by two factors: user privacy and service availability. Two alternatives could be considered: a centralized service, and a decentralized service running on mobile phones. However, in a centralized service (as in current OSNs) there are no incentives or appropriate business models to store for free the wealth of aggregated social data proposed in this work, and at the same time allow users full control over their data. Additionally, some OSNs institute particularly draconian policies concerning the ownership of user-contributed information and content. For example, users cannot easily delete their OSN-

- *Nicolas Kourtellis, Jeremy Blackburn and Adriana Iamnitchi are with the Department of Computer Science and Engineering, College of Engineering, University of South Florida, Tampa, FL, 33620. E-mail: {nkourtel, jhblackb}@mail.usf.edu and anda@cse.usf.edu.*
- *Cristian Borcea is with the Department of Computer Science, New Jersey Institute of Technology, Newark, NJ, 07102. E-mail: borcea@cs.njit.edu*

stored data (e.g., on Facebook) and they cannot export their social data to a competitive service in a transparent and easy way. Furthermore, users must trust their OSN provider for complying with privacy policies when exporting data to 3rd party applications, and not practicing Big Brother monitoring. In contrast, Prometheus gives users the final say on which peers in the system their private information is decrypted and used, and which applications and users are allowed to access it.

An alternative architecture stores social information on users' mobile devices, as proposed in [7]–[10]. Although much of the social information is nowadays generated by mobile devices, they are inherently unsuitable for running a complex social service such as Prometheus due to resource constraints: the mobile devices may not be always online or synchronized to support non-local inferences; and computational resources, and more importantly energy, are likely to be scarce.

An overview of the Prometheus service is presented in Section 2, and related work is covered in Section 3. Section 4 presents a detailed design of the service. We prototyped and evaluated Prometheus on a large-scale deployment on PlanetLab and tested its performance with high-stress workloads from emulated applications. The results in Section 5 show that the response time and system overhead for social inference execution is reduced when user social data are distributed onto peers using a socially-aware approach. In addition, in Section 7 we study the resilience of Prometheus to malicious attacks and demonstrate that its socially-informed design, as well as its social graph structure, make the system more resilient to attacks on the graph and inference requests. We also implemented CallCensor, a mobile social application that uses Prometheus to decide whether to filter out incoming calls based on the current social context. As shown in Section 6, the response time for this application running on a Google Android phone meets real-time deadlines, demonstrating that Prometheus' overhead is practical for real applications. We conclude in Section 8.

2 Overview

In order to better understand the functionality of the Prometheus service, we present it as part of the *social hourglass infrastructure* introduced in [11]. This infrastructure (illustrated in Figure 1) consists of five main components: (1) social signals, (2) social sensors, (3) personal aggregators, (4) a social knowledge service (SKS), and (5) social applications. Social signals are unprocessed user social information such as interaction logs (e.g., phone call history, emails, etc) or location and collocation information. Social sensors run on behalf of a user and parse the user's social signals, analyze them, and send processed social information to the personal aggregator of the user. The user's personal aggregator combines social information from the user's sensors and produces a personalized output based on user preferences. This personalized social information is sent for storage and management to a *social knowledge service (SKS)*. The SKS maintains an augmented social graph that can be mined by applications and services through an API that implements social inferences. Prometheus fulfills the role of the SKS in the social hourglass infrastructure.

2.1 Input from Social Sensors & Aggregators

We refer to *social signals* as the information that exposes social interactions between people. A vast diversity of social signals already exist as byproducts of Internet- or phone-mediated interactions, such as email logs, comments on blogs, instant messaging, ratings on user-generated content, phone call history, or via face-to-face interactions determined from (GPS or Bluetooth-reported) collocation data. For example, a social signal could reflect the interactions of two users over a soccer video posted on YouTube. These interactions reflect comments, "likes", other video sharing, etc.

Social sensors analyze users' social signals. Sensors are applications running on behalf of a user on various platforms such as their mobile phone, PC, web browser, or trusted 3rd party services. They transform the domain-specific interactions between the user (*ego*) and others (*alter*) into the following format:

$$ego :< alter, label, weight >$$

where $label$ specifies the interaction domain (e.g., $GoogleTalk$, $Facebook$, or $gaming$) and $weight$ associates a numerical value (between 0 and 1 in our implementation) to the intensity of the interaction, as in [12].

The *personal aggregator* is a trusted application typically running on a user-owned device (e.g., mobile phone) responsible for fusing and personalizing the information from the user's social signals. For each $alter$ from ego's social signals it sends to the SKS a tuple in the following format:

$$< ego, alter, new\_label, aggregated\_weight >$$

The aggregator can perform sophisticated analysis on the input from social sensors, for example, to differentiate between routine encounters with familiar strangers and interactions between friends [13]. It also applies user preferences for weighing social signals differently: for example, a Google Talk social signal might weigh less than Skype Chat in the aggregated value. Therefore, the aggregator can change both the labels and the weights received from social sensors into a quantitative representation of *ego* and *alter*'s social interactions.

2.2 Prometheus as a Social Knowledge Service

The SKS provides a mechanism for storing and managing user social data and exposing them to applications and services. Prometheus, which fills-in the role of SKS in the social hourglass infrastructure, distributes the social information received from the personal aggregators on multiple user-contributed peers for better service and data availability, while protecting access to data via encryption and user-specified access policies.

Prometheus design supports the following functionalities:

Persistent service for mining the social graph is achieved through employing a p2p architecture on user-contributed resources without the availability constraints of architectures

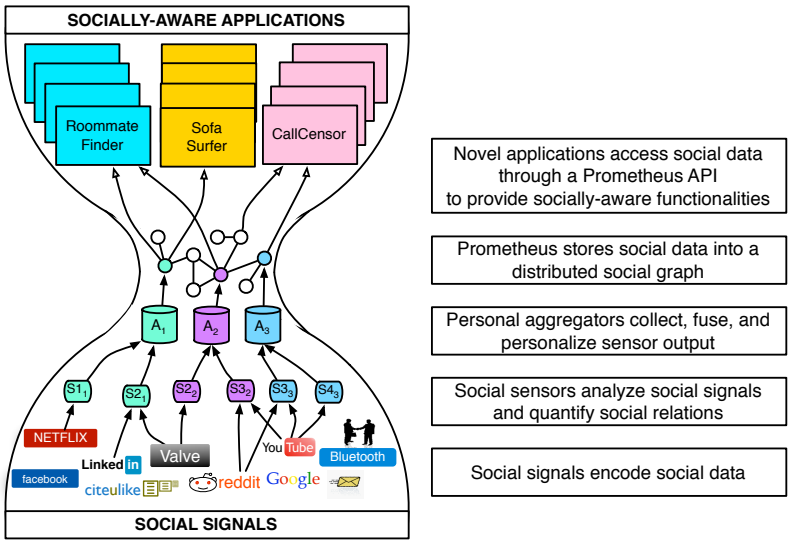


Fig. 1. Prometheus in the social hourglass infrastructure [11].

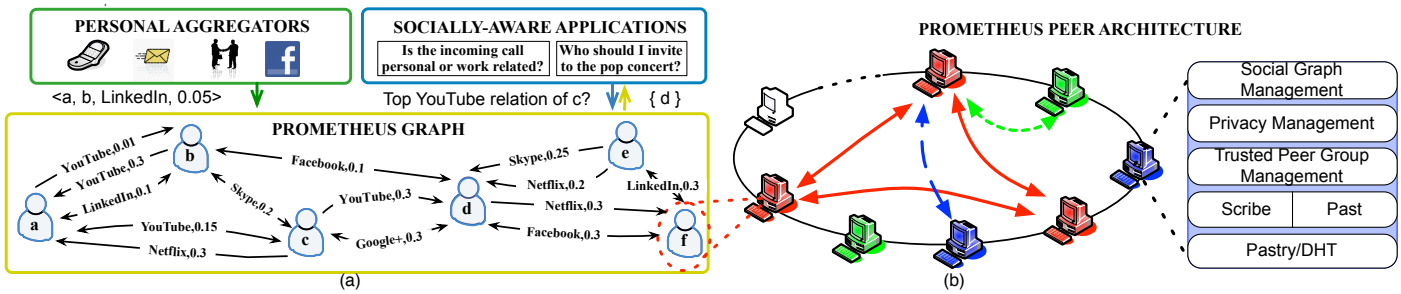


Fig. 2. (a) Social sensor input stored as a directed, labeled, weighted multi-graph. Applications mine the graph via Prometheus API social inferences. The social graph (left) is distributed on the P2P infrastructure (right). (b) Prometheus peers organized using Pastry, a DHT-based overlay, Scribe, a DHT multicast infrastructure, and Past, a DHT storage system. Same color machines comprise a user's trusted peer group allowed to decrypt and mine the user's social subgraph. Continuous (*red*) arrows show communication for social graph and group maintenance. Prometheus inferences are executed between peers in a decentralized fashion (dashed arrows).

based on mobile phones (e.g., [7]–[10]). In particular, our objective is to allow applications access to distant nodes in the social graph, such as friends-of-friends queries.

Support to an unrestricted set of social applications and services is provided via two mechanisms: an API that exports social inferences (such as social strength between indirectly-connected users) and fine-grained social information represented by social edge labels and weights. For the latter, Prometheus maintains a weighted, directed, labeled, and multi-edged social graph, where vertices correspond to users and labeled edges correspond to interactions between users, as reported by their aggregators (Figure 2(a)). Edge labels specify the type and weights specify the intensity of the interaction between users. Edge directionality reflects the asymmetry of users' perception of a social relationship [14] and allows for potentially conflicting reports from the aggregators of the interacting users.

User-controlled data access is guaranteed via two mechanisms. First, data owners can specify access control policies to limit access to their own information. Second, all data is encrypted; however, for mining the social graph information, some peers need to decrypt data in order to process it. Thus, each user selects a group of *trusted* peers; only these peers can decrypt the user's social data.

In addition, Prometheus deals with the typical challenges of churn-prone p2p systems by leveraging well-established, previously proposed solutions in distributed hashtable overlays (DHTs). Figure 2(b) presents an overview of the Prometheus architecture. Each peer runs three software components: 1) for social graph management; 2) for privacy management; and 3) for trusted peer group management.

2.3 Output to Applications and Services

Prometheus exposes an interface to a rich set of social inference requests computed over the distributed social graph. For example, an application can request on its user's behalf to receive the top n relations with whom the user interacts over Facebook. Similarly, it can request the social strength between a user and another user not directly connected in the social graph. Prometheus provides the mechanism with which inference requests can access not only a single user's social neighborhood (i.e., directly connected users), but also

the social data of users located several hops away in the global social graph. As mentioned before, all inferences are subject to user-defined access control policies enforced by the trusted peers of the data owner.

2.4 Prometheus in Use: CallCensor Application

An application scenario is presented below for explaining how Prometheus is used in the context of the social hourglass infrastructure. Let us assume that user Bob installs a new context-aware phone-call filtering application (e.g., CallCensor [15]), that allows him to filter incoming calls from another user $Alice$, based on (1) their declared professional relationship on LinkedIn, (2) their personal relationship on Facebook, (3) the phone call interactions between them, (4) their chatting activity over Google Chat and Skype, and (5) Bob's current location and collocation with other individuals.

At installation, the application checks with Bob's aggregator which of the required and optional social sensors he is currently registered with. These sensors report Bob's activity to Prometheus, subject to a personalized filter stored and applied by his aggregator (as mentioned earlier). This filter is updated rarely—when there are significant changes in activity patterns or new social sensors are deployed—or can be left to a default setting that weighs all signals equally. CallCensor will mine Prometheus' aggregated social information on Bob in order to decide which calls to let through. Thus, personal calls could be automatically silenced during professional meetings, but co-workers' or other professional-related calls will be let through.

To make this decision, CallCensor will query Prometheus for a list of Bob's social contacts that are also geographically close to Bob, allowing the application to infer his social context. LinkedIn social edges can be used to infer work-related contacts. Further, CallCensor will query Prometheus for the type of connections between the caller and Bob, and their social distance. Calls from contacts classified as "friends of friends" (i.e., 2-hop social or work neighborhood) could be let through, depending on user-specified application-related preferences. For instance, the application might allow Bob to specify that calls from "co-workers of co-workers" connected via LinkedIn ties will be let through, if also supported with high weight of phone call and Skype interactions.

If not all necessary social signals are available for Bob, they will be identified when the application is first installed. An out-of-band service lists the various implementations of sensors and their social signals. Bob will be prompted to agree with the deployment of missing sensors. His aggregator, as his personal assistant, provides the credentials for these sensors (e.g., the Facebook password to access wall posts). Sensors are deployed on where each social signal is (e.g., as a Facebook application) or on user-controlled platforms (e.g., a web browser sensor running on user's desktop). The cognitive load on the user is determined by the level of sophistication desired for social inferences, from default, one-size-fits-all settings to fully personalized.

3 Related Work

Decentralized management of social data using peer-to-peer architecture has been proposed in other studies such as PeerSoN [16], Vis-à-Vis [17], Safebook [18], LotusNet [19] and LifeSocial.KOM [20].

PeerSoN [16] is a two-tier system architecture: the first tier is used to lookup (through a DHT) for metadata of users, current location/IP, notifications, etc; the second tier allows opportunistic and delay-tolerant networking and is used for the actual contact between peers and users and the exchange of data (files, profiles, chat, etc). Users can exchange data either through a DHT storage or directly between their devices. Vis-à-Vis [17] introduces the concept of a Virtual Individual Server (VIS) where each user's data are stored on a personal virtual machine. While similar to the trusted peer concept in Prometheus, VISs are hosted by a centralized cloud computing provider to tackle peer churn, whereas Prometheus provides a truly decentralized infrastructure and uses social incentives to reduce churn of user-contributed peers.

Safebook [18] is a decentralized OSN with a 3-component architecture: a trusted identification service to provide each peer and user unambiguous identifiers, a P2P substrate for lookup of data, and matryoshkas, concentric rings of peers around each member's peer that provide trusted data storage, profile data retrieval and obscure communication through indirection. Profiles of users in Safebook are replicated in the innermost layer of a user's matryoshka to increase availability of the data. Similarly, Prometheus users employ multiple trusted peers and forwards messages between peers based on social relations and inference requests, while restricting a large scale view of the graph through P2P decentralization. Prometheus differs, however, in the exposure of the graph as a first class data object through inference functionality.

LotusNet [19] is a DHT-based OSN that binds user identities with their overlay nodes and published resources for increased security. User data such as social information, content, etc, are stored encrypted on a DHT. Users control access to their data by issuing signed grants to other users applied during retrieval of data. Services such as event notification, folksonomic content search and reputation management are demonstrated. In LifeSocial.KOM [20], user information is stored in the form of distributed linked lists in a DHT and is accessible from various plugin-based applications, while enforcing symmetric PKI to ensure user-controlled privacy and access. However, the data of a user are isolated from other users and peers access them individually. Commercial efforts such as [21]–[23] implement distributed social networking services and allow users to share content in a decentralized fashion.

Our work differs from the previously noted academic and commercial approaches in the following ways. First, it not only collects and stores user social information from multiple sources in a P2P network, but also exposes social inference functions to socially-aware applications that mine the social information stored. Second, it addresses the space between centralized and fully decentralized systems, as it allows multiple users to store social information on common peers they trust. These peers can locally mine the collection of social data entrusted to them by the groups of (possibly socially-connected) users, thus improving social data access and quality of inferences to applications. Third, Prometheus enables users

to select trusted peers independent from the users' (mobile) devices to decrypt and mine their social data at any time, thus increasing service availability.

4 DESIGN

Prometheus manages a labeled, weighted social multigraph distributed on a collection of user-contributed peers. Unlike other p2p solutions, Prometheus does not require each user to contribute a peer. Instead, it leverages social incentives to allow a user's data be managed by the peer contributed by a trusted social relation. This approach embeds social awareness in the service design and has three immediate benefits. First, it improves scalability with the number of users (as the size of the p2p system can be much smaller than the number of users in the system). Second, socially-incentivized users keep their computers online, thus reducing churn [2], [3] and consequently increasing service availability. And finally, because socially-related users are likely to select the same trusted peers for their social data management, the resulting collocation of socially-connected users decreases remote access for social graph traversal. A less intuitive benefit from the social-based mapping of the social graph onto the p2p network includes a reduced vulnerability to malicious attacks, as demonstrated in Section 7.

As presented in Section 2, the data stored in Prometheus consist of labeled and weighted social edges as reported by personal aggregators. Data are stored encrypted in Past [24], a DHT-based storage, subject to the DHT mapping function of the Pastry overlay [25]. The trusted nodes are capable of decrypting their user data for graph processing. Prometheus uses a public-key infrastructure (PKI) to ensure both message confidentiality and user authentication. Users have public/private keys for both themselves and their trusted peers decrypting their social data.

4.1 User Registration

A user registers with Prometheus by creating a uniform random UID from the circular 128-bit ID space of the DHT (typically the hash of their public key). At registration time, the user specifies her peer(s) contributed to the network (if any). The user then creates a mapping between her UID and the list of these peers' IP addresses and signs it with her private key for verification. By contacting any peer in the network to momentarily join the DHT ring, the user stores this mapping in the network as the key-value pair $UID = \{IP_1, ..., IP_n\}$. When one of these peers returns from an offline state, it updates the mapping with its current IP address.

The user also compiles a list of other Prometheus users with whom she shares strong out-of-band trust relations and searches the DHT storage for their machine mappings (using their UIDs). From the returned list of peers owned by these users, she selects an initial set as her trusted peer group. The larger this set, the higher the service availability; at the same time, the consistency and overall performance may decrease. The trusted peer group may change over time, as shown next.

4.2 Trusted Peer Group Management

The main issues concerning the trusted peer group management are group membership management and search for trusted peers. For the trusted peer group management we leverage Scribe [26], a group communication solution designed for DHTs.

A user adds peers to her trusted peer group by initiating a secure three-step handshake procedure to establish a two-way trust relationship between her and a peer owner. During this handshake process, the following steps take place: 1) the user sends an invitation to the peer owner, 2) if the peer owner trusts the user not to be malicious, it replies with an acceptance message, 3) upon acceptance of the invitation, the user sends to the peer owner the group keys to enable the peer to join her trusted peer group. Upon receiving the group keys, the new peer sends a *subscribe* request to the Scribe trusted group of the user. The group's handle is the concatenation of the predefined string "Trusted_Peer_Group" and the user's *UID* and can be used to publish signed messages to the group using a multicast protocol.

To remove a peer from the trusted peer group, the user creates new group keys and distributes them via unicast to the still trusted peers. She also re-encrypts her social data and replaces the copy in the DHT storage for future use. This removal of the peer is *multicasted* to all group peers and the peer is unsubscribed from the group. The distribution of new keys and re-encryption of the social data disallows the removed peer from decrypting updated versions of the user's social data in the future. If a peer owner decides to remove her peer from a trusted group of another user (e.g, due to overload or suspected malicious activity from the user), the removal request is multicasted to the group, which alerts the data owner to execute the same procedure as above.

The social graph for a user is unavailable if all her trusted peers leave the network; no service requests involving this user can be answered until a trusted peer rejoins the network. However, we ensure durability of users' generated data (i.e., input from social aggregators) via encrypted storage and replication in the DHT by Past.

Service requests for a user can be sent to any peer, but only the user's trusted peers can decrypt and use the user's data. A requesting peer can find a user's trusted peers by submitting a *multicast* request with handle *Trusted_Peer_Group_UID*. With the multicast, all online group trusted peers are required to respond with their IP and signed membership, which is verified for authenticity with the user's group public key. The requesting peer creates a trusted peer list (TPL) of IPs based on peer responses. The multicast allows the requesting peer to have peer alternatives in case of churn or erroneous communication with the first responding peer.

The search for trusted peers follows the Scribe multicast tree which utilizes the Pastry DHT routing and its network proximity metric to forward messages to near-by peers, thus reducing overall delays. In effect, the responds from trusted peers are sorted in the TPL by overall latency. The peer, upon creating the TPL for a user's group, can communicate directly with the individual trusted peers, preferably the one responding

fastest. Prometheus caches the TPL after the first access and updates it when the trusted peers are unresponsive, changed their IP, or became untrusted. Users could also apply their own policies for refreshing the local TPLs based on their usage patterns (e.g., daily).

4.3 Distributed Social Graph

Prometheus represents the social graph as a directed, labeled, and weighted multi-edged graph [27], maintained and used in a decentralized fashion as presented in Figure 3.

Multiple edges can connect two users, and each edge is labeled with a type of social interaction and assigned a weight (a real number in the range of $[0, 1]$) that represents the intensity of that interaction. The labels for interactions and their associated weights are assigned by the personal aggregator of each user. From an application point of view, distinguishing between different types of interactions allows for better functionality. For extensibility, we designed Prometheus to be oblivious to the number and types of social activities reported by the aggregators. Furthermore, we chose to represent the graph as directed and weighted because social ties are asymmetrically reciprocal [14]. This representation also limits illegitimate graph uses (e.g., for spamming). The latest known location of a user and an associated timestamp are maintained as an attribute of the user's vertex in the graph.

The social data for each user are encrypted, signed and stored in Past in the append-only file $Social\_Data\_UID$. Only the user's personal aggregator can send updates to create new edges, remove old edges, or modify edge weights. These updates are appended in this file as records with a sequence number. Only the user's trusted peers can decrypt and use these records.

Trusted peers periodically check the file for new records and retrieve all such records: this is easily done based on sequence number comparison starting from the end of the file. Each trusted peer decrypts the new records and verifies their authenticity. Then, it updates the local subgraph of the user with the newly retrieved records. Since the file is append-only, the trusted peers can access it at any time: in the worst case, they will miss the latest update. Therefore, for short periods of time, the trusted peers may have inconsistent data, but this is not a major problem as social graphs do not change often [28].

Edges may "decay" over time if few (or no) activity updates are received [29]. This aging process should be activity specific, but it should also reflect the user's social habits and interests: users who are less socially active and users with a great number of friends should have their relationships age slower. Currently, the system applies a simple aging function to reduce an edge's weight by 10% for every week the two users do not interact over the particular label (thus, the connection never completely disappears and the aging happens slowly). A user's aggregator may specify a different decrement value of the weight and the time period for aging (these values are also stored in the $Social\_Data\_UID$ file for each user).

4.4 Social Inference API

Prometheus exposes to applications an API of social inference functions that are executed in a decentralized fashion; more complex inferences can be built on top of this set.

Relation_Test(ego, alter, α, χ) is a boolean function that checks whether ego is directly connected to $alter$ by an edge with label α and with a minimum weight χ. The CallCensor application can use this function to determine if an incoming call is from a coworker with a strong social tie, and therefore, should be let through even on weekends.

Top_Relations(ego, α, n) returns the top n users directly connected to ego by an edge with label α, and ordered by decreasing weights. An application can use this function, for example, to invite users highly connected with ego to share content related to activity α.

Our previous study [15] revealed volatility of the peer-to-peer communication and long response delays during multi-hop inference execution. Thus, we redesigned the following API functions to offer better quality of service to applications, by allowing them to define not only inference-specific parameters (such as label and weight), but also a new a timeout parameter T, which limits the application waiting time.

Neighborhood(ego, α, χ, radius, T) returns the set of users in ego's social neighborhood who are connected through social ties of a label α and minimum weight χ within a number of social hops equal to *radius*. The *radius* parameter allows for a multiple hop search in the social graph (e.g., setting *radius* to 2 will find ego's friends of friends). The CallCensor application can use this function to determine if a caller is in *ego*'s work neighborhood in the social graph even if not directly connected.

Proximity(ego, α, χ, radius, distance, timestamp, T) is an extension of the neighborhood function which filters the results of the neighborhood inference based on physical distance to ego. After the location information is collected for ego, and $neighborhood$ returns a set of users, $proximity$ returns the set of users who are within $distance$ from ego and their location information is at most as old as $timestamp$. Users who do not share their location or have location information older than the $timestamp$ are not returned. A mobile phone application might use this function to infer the list of collocated coworkers within a certain distance of ego.

Social_Strength(ego, alter, T) returns a real number in the range $[0, 1]$ that quantifies the social strength between ego and $alter$ from ego's perspective. Past studies in sociology [30] observed that two individuals have meaningful social relationships when connected within two social hops (i.e., "horizon of observability"), and their relationship strength greatly depends on the number of different direct or indirect social paths connecting them. Therefore, the two users in this function can be directly or indirectly connected and through multiple parallel social paths (e.g., paths $c \rightarrow b \rightarrow e$ and $c \rightarrow d \rightarrow e$ between c and e, Figure 3). The return value is normalized, as explained below, to ego's social ties, to ensure that the social strength is less sensitive to a particular social activity of the users.

Assume $\Lambda_{i,j}$ is the set of labels of edges for two directly connected users i and j, $w(i, j, \lambda)$ is the weight of an edge between i and j over label λ, and Θ_i the set of directly connected neighbors to i. Then $nw(i, j)$ is the overall normalized

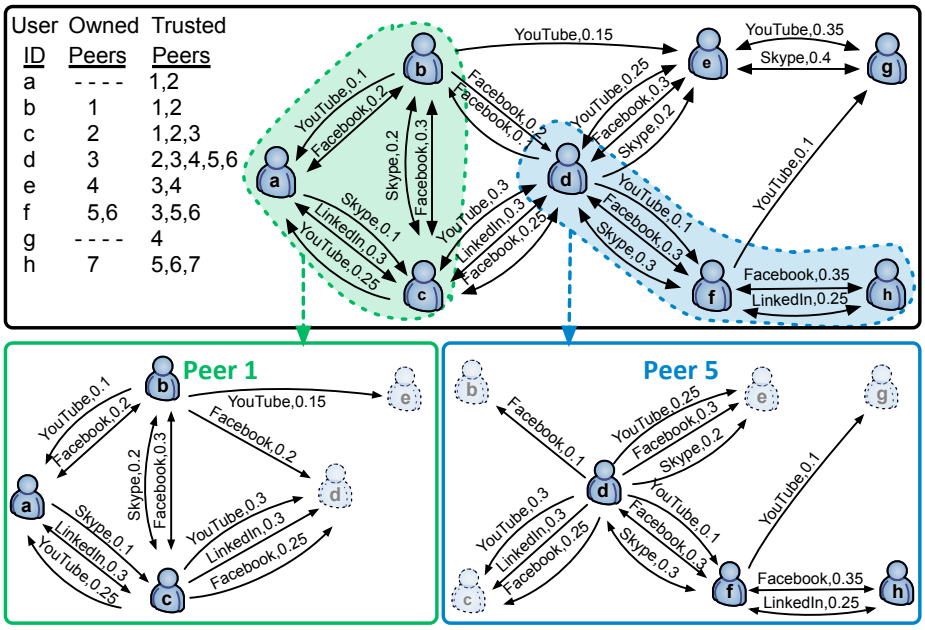


Fig. 3. An example of a social graph for eight users (*a-h*) distributed on seven peers. The top figure shows the mapping between users, peer owners, and trusted peers (upper left corner) and how users are connected with each other over social edges, each marked with its label and weight. The bottom figures illustrate the subgraphs maintained by peers *1* and *5*. Users in dark color (e.g., a, b, c on peer *1*) trust the peer to manage their social data. Users in light-shaded color (e.g., e, d on peer *1*) do not trust the peer but are socially connected with users who do.

weight between i and j:

$$nw(i,j) = \frac{\sum_{\forall \lambda \in \Lambda_{i,j}} w(i,j,\lambda)}{\max_{\forall j \in \Theta_i} \left(\sum_{\forall \lambda \in \Lambda_{i,j}} w(i,j,\lambda) \right)} \quad (1)$$

Also, assume $\Gamma_{i,m}$ is the set of different 2-hop paths between indirectly connected users i and m. Then $S(i,m)$ is the return value for social strength between i and m, over a multi-level 2-hop path:

$$S(i,m) = 1 - \prod_{\substack{\forall p \in \Gamma_{i,m} \\ \forall j \in \Theta_i \cap \Theta_m}} \left(1 - \frac{\min\{nw(i,j), nw(j,m)\}_p}{2} \right) \quad (2)$$

Such a function could be used, for example, to estimate social incentives for resource or information sharing.

Inference Function Execution: An application can submit a social inference request for a user x to any Prometheus peer in a secure fashion (i.e., encrypted and signed). The receiving peer creates x's TPL (as explained in Section 4.2) and forwards the request to the fastest responding trusted peer of x, which verifies the submitter's identity and enforces x's access control policies (described next). The peer fulfills the request by (1) traversing the local social subgraph for the information requested by the application, (2) encrypting and signing the result, and (3) returning the result to the application.

For functions that can traverse the graph for n hops (i.e., *Neighborhood* and *Social_Strength*), the peer submits encrypted and signed secondary requests for information about other users to their trusted peers, as follows. A secondary request includes the UID of the original submitter (in order to verify her access rights). A time period of $T*(n-1)$ seconds is given to the secondary peers to respond with their results (peers at each *hop* use independent clocks for T seconds). Each receiving secondary peer authenticates the request and checks the access control policies for the requesting user. If the request is granted, the result is returned to the requesting peer. If the request still needs more information, that peer (e.g., at hop k) repeats the same process and submits a secondary request with an adjusted $T*(n-k-1)$ timeout. Finally, the original requesting peer recursively collects all the replies and submits the final result to the application.

4.5 Access Control Policies

Users can specify access control policies (ACPs) upon registration and update them any time thereafter. These policies, stored on each of the user's trusted peers, are applied each time an inference request is submitted to one of these peers to access social information for the particular user. For availability, the policies are encrypted, signed and stored in the DHT, allowing a rejoining trusted peer to recover updated policies. The same mechanism used for updating the social graph is used to update policies. As future work, we plan to investigate the provision of strong consistency and conflict resolution for policies.

ACPs are comprised of two parts: the social data object(s) to be accessed and the specification(s) to be met before access is granted for the particular data object(s). They are defined as entries of the ACP_UID file in the following format:

<Social Data Object(s)> :: <ACP Specification(s)>

Table 1 presents a list of such access control policy definitions. By design, ACPs are whitelists. Any of the specifications can be used to allow access to any of the data objects. Each of the data objects, as well as each of the specifications, can be combined with logical operators (e.g., *AND*, *OR*, *NOT*, etc.), to create more complex access policies. To verify the access rights, Prometheus may call its inference functions, when applicable. For example, to detect whether the request's originating user is within n-hops, she is checked against the result of an n-hop *neighborhood* inference. ACPs also allow for blacklisted users (and their peers) for convenience.

Figure 4 shows an example of the set of access control policies for user Bob. Bob allows *work*-related social information to be given to requests coming from users within 2 hops who are connected with him over *LinkedIn* label, or when these requests come from the *CallCensor* application. Also, he allows his parents and his brother to access his exact location at any time. If a neighborhood inference request for the *Facebook* label is submitted to Bob's trusted peer, Prometheus checks his ACP in the order *Blacklist→labels→weights*. Users $Alice$ and $Gary$ and $Alice's$ peer are excluded from all types of inferences and cannot receive any information about Bob.

TABLE 1
Access Control Policy Definitions.

Social Data Object(s)		ACP Specification(s)	
Social edge label	α	Social distance	ρ
Social edge weight	χ	Social edge label	γ
User location	Δ	Social edge weight	y
		Originator user	B
		Originator peer	P
		Intermediate user	C
		Intermediate peer	M
		Application	S
		Originator's location	L

$< \chi = 0.3 >::< \rho = 1$ AND $S = SofaSurfer >$
$< \Delta >::< B = mom$ OR $B = dad$ OR $B = brother >$
$< \alpha = Skype >::< \rho = 2$ AND $\gamma = Skype$ AND $y = 0.2 >$
$< \alpha = Facebook$ AND $\chi = 0.2 >::< \rho = 1$ AND $\gamma = Facebook >$
$< \alpha = LinkedIn >::< (\rho = 2$ AND $\gamma = LinkedIn)$ OR $S = CallCensor >$

$< blacklist >::< B = Alice$ OR $B = Gary$ OR $C = Alice >$

Fig. 4. Example set of Access Control Policies of Bob.

To protect their data from illegitimate access, users can set strict ACPs, for example, serving only requests originating from trusted 1-hop friends. Such ACPs will disallow 2-hop contacts from accessing any data, but will also restrict the usability of multi-hop inferences such as $social\_strength$ or $neighborhood$. Overall, we anticipate a tradeoff between how ACPs are defined to protect social data, and the usability of the platform through social inferences on these data. However, as demonstrated in the ACPs definition, users can fine-tune them to allow different access rights to particular sets of users, enabling inferences, and in extend applications, to maintain their functionality by accessing specific portions of user data.

5 Performance Evaluation

We implemented Prometheus on top of the FreePastry [31] Java implementation of Pastry DHT which also provides API support for Scribe and Past. The Prometheus prototype was deployed and evaluated on PlanetLab (PL). We performed various optimizations and fine tuning from our earlier work [15] to better handle the communication volatility and peer churn typically observed in a distributed infrastructure. We also redesigned the API as it required applications and peers to exchange string-formatted messages with no extensibility on the number of request fields. The new API allows applications to submit a serializable class-based request by defining within the class object not only inference-specific parameters (such as label and weight) but also the new timeout parameter, which declares the application waiting time per hop when requesting multi-hop inferences. Furthermore, applications and peers can define parameters such as request id, timestamp and error flags. Overall, the new API design enhanced extensibility for future inference parameters, portability across different platforms, and improved the peer-to-peer and peer-to-application communication during inference execution.

5.1 Experimental Setup

We executed two sets of experiments to evaluate the service performance. The three metrics used in this evaluation were: (1) *end-to-end response time* to quantify the application-perceived performance, (2) *percentage of completion* to quantify the tradeoff between request response time and level of request completion, and (3) *number of network messages* to quantify the service overhead.

For these experiments we deployed Prometheus on 100 selected PL peers worldwide. While we did not apply a controlled peer churn, the peers were subjected to a PL churn of an average of up to 5%.

For the first set of experiments, we set the application timeout to 15 seconds per social hop.

This timeout is needed to deal with communication delays due to long RTTs (200–300ms on average), busy network interfaces (the PL peers were shared with other researchers), uncontrolled peer churn, and forwarding of requests. Prometheus uses this parameter to decide when to aggregate the intermediate results received so far and send them to the requesting peer. We varied this timeout in the second set of experiments.

To remove any bias from the users' data placement on failing or slow-responding peers and the submission of requests from or to such peers, each peer submitted application workload on behalf of all the social graph's users and we report averaged results. Moreover, we assumed users defined ACPs that allowed access to all their data from other users, thus stressing the system at maximum load.

5.1.1 Decentralizing the Social Graph

We used a bidirectional graph of 1000 users created with a synthetic social graph generator described in [32], which consistently produces graphs with properties such as degree distribution and clustering coefficient similar to real social graphs. Edge weights were initially set to 0.1 and were dynamically updated over time based on an empirically driven model derived from [33], as explained in Section 5.1.2.

We opted for a synthetically generated social graph in order to assess Prometheus performance independent of the particularities of specific real social graphs. The graph size was chosen to reflect a realistic scenario of distributing a social graph on a 100 friend-contributed peers network: each contributing user is trusted by a relatively low number of relations. We vary this number from 10 to 50 (allowing replication). However, we must note that our previous study [34] showed that one peer can easily handle requests for 1000 users with no performance penalty. The scale of the social graph and consequently of the P2P network tested is sufficient for evaluation of meaningful social inferences typically traversing a small social neighborhood of up to 2 hops. Therefore, 1000 users mapped on 100 peers provided enough variation for statistically significant measurements. Larger scale testbeds and graphs would evaluate DHT routing performance, which has already been studied in the past and is not the focus of this work.

We distributed the social graph on peers using a random and a social mapping. In the random mapping, users' social data are stored on randomly selected peers. Consequently, groups of random users are mapped on the same peer. The social mapping corresponds to a more realistic scenario, where a group of socially-connected users share the resources provided by a peer contributed by a group member. We created such a social mapping using a modified version of the community detection algorithm introduced in [35] that allowed us to control the number of communities and their average size. The algorithm takes as input a social graph, the number of communities to be identified (which in our case is the number of peers in the system) and the minimum acceptable community size. The algorithm iteratively removes the social edge with the highest edge betweenness centrality if by removing it a new community of the desired size is created. Removal of edges continues until the specified number of communities is met. Users from a community are then mapped on the same peer.

However, even in the social mapping, neighboring communities in the social graph where mapped on random peers worldwide. This setup allows geographically-close and socially connected communities to store their information on random peers across the globe, which enables us to examine a worst case network performance of the platform.

The average number of users mapped per peer received values N=10, 30 and 50 users/peer. The number of PL peers was kept constant to 100, forcing the user groups to overlap for N>10. This resembles the realistic scenario of overlapping social circles with some users participating in more than one circle (peer), and thus having multiple trusted peers. In effect, user data were replicated on K=N/10 peers on average, hence K=1, 3 and 5 trusted peers/user.

5.1.2 Synthetic Workloads

We emulated the workload of a social sensor and two social applications based on previous system characterizations [33], [36], [37]. The emulated sensor tested the platform's ability to manage and incorporate new social input under high-stress load. The emulated user social applications tested the end-to-end performance of the *neighborhood* and *social_strength* requests.

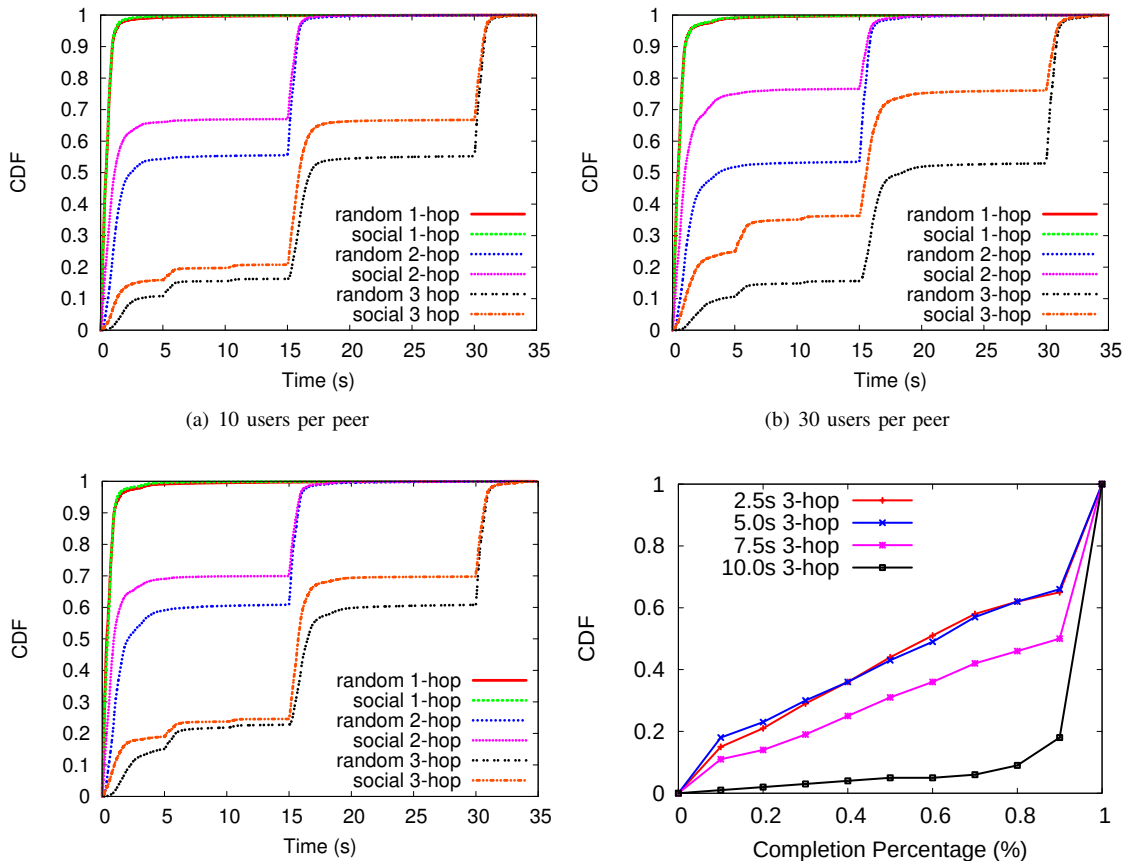
Social edge weight update: We emulated a Facebook social sensor based on a Facebook trace analysis [33]. The workload was characterized by the probability distribution function for users to post comments on walls and photos. Users were ranked into groups based on their social degree and each group was mapped onto a probability class using the cumulative distribution function from Figure 8 in [33]. To emulate a social interaction from ego to $alter$, a group was selected based on its associated probability, and a user ego from the group (not selected yet) was picked as the source of input. $Alter$ was randomly selected from ego's direct social connections. The weight of each input was kept constant to 0.01 for all users. Since users were picked based on their social degree, users with higher degree probabilistically produced more input, leading to higher weights on their corresponding edges in the graph.

Workload for *Neighborhood* Inference: A neighborhood request is a limited distance flood in the social network, similar to a tweet in Twitter. We used a Twitter trace analysis [36] to associate a tweet with a neighborhood request (centered at the source of the tweet) in Prometheus. Thus, we extracted the probability distribution function of submitted requests. Users were ranked into groups based on their social degree. Using Figure 4 in [36], each group was mapped onto a particular probability to be selected and submit a neighborhood request. Once the group was selected, a user (not selected yet) was picked to be the source of the request. The number of hops for the request was randomly picked from 1, 2 or 3 hops and the weight was randomly picked, with an upper bound of 0.1 to produce maximum inference request load in the system.

Workload for *Social_Strength* Inference: We used an analysis of BitTorrent traces to emulate the workload of a battery-aware BitTorrent application [38] on mobile devices: a user may rely on social incentives to be allowed to temporarily "free ride" the system when low on battery. Members of the same swarm check their social strength with the needy leecher to see if they want to contribute by uploading on her behalf. We assumed that users participated at random in BitTorrent swarms. Two users were randomly selected as the source and destination of the social strength inference request. The source user was associated with a total number of requests she would submit throughout the experiment. This number was extracted from an analysis of BitTorrent traces (Figure 9b in [37]).

5.2 End-to-End Performance

The first set of experiments had two goals: (1) to measure Prometheus' performance over a widely distributed network such as PL, and (2) to assess the effect of socially-aware trusted peer selection on the system's overall performance. For



(a) 10 users per peer

(b) 30 users per peer

(c) 50 users per peer

(d) 3-hop timeout

Fig. 5. (a)–(c) CDF of the average end-to-end response time of the *neighborhood* inference for the *random* and *social* mappings, different social hops and number of users per peer. The distributions of the *social* mapping were found statistically different from the *random* mapping using the Kolmogorov-Smirnov test and for $p < 0.0001$. (d) CDF of the average completion percentage of 3–hop neighborhood requests for varying timeout values.

every run, more than 1 million *social_strength* and *neighborhood* requests and more than 100 thousand social input updates were submitted from the emulated applications and social sensor. Figures $5(a)$–(c) show the cumulative distribution function of the end-to-end average response time for the *neighborhood* inference for different social hops and number of users per peer. We do not include the results for the *social_strength* inference as its performance is almost identical to the one for *neighborhood* for 2 hops, since this function verifies all the possible 2-hop paths between two users.

From these experiments we observed that the response time was relatively high due to the overloaded testbed, especially for 2 and 3 hops. In our worldwide PL testbed, the average RTT was 250ms. In addition, the testbed was generally loaded with other projects running along with ours, leading to about 1–2 seconds delay just to establish a reliable TCP connection to submit a request to a peer or receive an inference result from a peer. Furthermore, the response time greatly depended on the number of hops to be traveled by the request and the number of peers to be contacted at each hop. Thus, even though the aggregation of a request result is executed in parallel per social hop, multiple peers must be contacted per hop, with potentially long response delays to complete a request at the highest rate possible. For example, for 10 users/peer in the social (random) mapping, an average of 37 (48) peers have to be contacted to collect *neighborhood* results from users located 3 social hops away from the source user (the number of users returned is $\sim$350). In our experiments, such a 3-hop request took about 30 seconds to reach $\sim$100% completion rate with an enforced 15-second timeout/hop.

Moreover, the response times were dominated by the network delays observed in PL (tens of seconds) in comparison to the overhead for different local activities such as request parsing, signature verifications and subgraph traversal on each peer (tens of milliseconds).

The overloaded testbed also led to increased communication delays while creating trusted peer lists of users. Since a request for a user x can arrive at any random peer, this peer has to first create x's TPL—an operation which involves several time-consuming lookups in the DHT and results in multiple peer traversals—and then forward the request to the first trusted peer to respond. The distribution of the overhead associated with the cold start of creating a user's TPL over the PL infrastructure had a 50^{th}, 90^{th} and 99^{th} percentile of 1.05, 2.15 and 8.78 seconds, respectively. Thus, for the majority of the users this process can be fast, but for some users it can take

as much as 8–10 seconds. Similar delays (6–10 seconds) were reported in [17] for the group activities of the Vis-a-Vis system running over PL nodes. To mitigate this problem, Prometheus caches the TPL as explained in Section 4.2. The plots in Figure 5 show the performance using this caching mechanism, as we are interested in testing Prometheus' performance in inference execution and not in DHT lookups.

From these experimental results we draw the following lessons on Prometheus' performance.

Lesson 1: *The social-based mapping of users onto peers improves by* 20-25% *the request response time and reduces by* 20–40% *the system message overhead.* In the cases of 10 and 30 users/peer (Figure 5(a)-(b)), gains of up to 20-25% in response time over the random mapping are observed. The difference is more visible for 2 and 3 hops, as the 1 hop function is computed either locally at the submitting peer or the first available trusted peer of the source user. For the case of 50 users/peer (Figure 5(c)), the system continues to perform better with the social than the random mapping, but the improvements are smaller than between 10 and 30 users/peer. This is because under this value, the average number of peers that must be contacted for information is reduced in both mappings, but the reduction is more prominent in the random mapping. The average number of messages in the system (not shown here) is reduced by ∼20–40% with the social-based mapping in comparison to the random-based mapping.

Lesson 2: *Increasing the service availability (through data replication) and number of users per peer does not always improve the end-to-end response time and message overhead.* In general, since inference requests for a user can be fulfilled by any of her trusted peers, we observe that increasing the availability of users' data by a factor of 3, and correspondingly increasing the number of users mapped per peer, improves the end-to-end performance by up to 25% and reduces the message overhead in the system by ∼30%. This overall performance gain is due to the following two reasons. First, having more user data on each peer allows for more requests to complete with fewer network peer hops. Second, given high peer churn and vulnerable P2P communication, more service availability per user means more alternative trusted peers to contact for an inference request to be fulfilled faster. However, when applying K=5 and N=50, we observe a reduction in the performance improvements due to increased workload on each peer from the additional users mapped.

Lesson 3: *Caching inference results and geographic social graph decentralization can improve scalability and response time by a factor of* 10. By placing socially-close communities on random peers, we examined a pessimistic experimental scenario of longer than expected delays for request execution. However, in reality, we expect neighboring communities in the social graph being placed in geographically close peers (e.g., same country peers) instead of random. This effectively reduces delays by one order of magnitude, since the average (median) RTT between PL peers of the same country was 25.5 (15.7) msec in our experiments for over 35 countries. To further reduce execution delays, we plan to implement caching of recently computed results as well as pre-computing results in the background. These methods are expected to work well as the social graph rarely changes [28] and will allow the inference execution to scale easier to thousands of peers.

5.3 Response Time vs. Completion Rate

The longer an application is willing to wait, the more complete the information returned by the social inference is. The previous experiments confirmed that a long timeout of T=15 seconds per hop offers a high completion rate. We designed a second set of experiments to measure the trade-off between end-to-end response time and response completion rate. We varied the timeout T to $2.5, 5.0, 7.5, 10.0$ seconds and used the social mapping with 30 users/peer.

Figure 5(d) shows the cumulative distribution function of the average completion percentage of the 3-hop neighborhood requests under different application-set timeout values. The results for 1 and 2 hops showed practically 100% completion for all requests and timeout values and are omitted for brevity. Overall, we observe a clear trade-off between request completion rate and application waiting time for response. For example, when T=7.5(10)seconds, about 50%(80%) of the requests have more than 90% completion. A real-time social application—e.g., "using the *proximity* inference, invite my 2-hop football contacts for celebration of the team's victory"—could set a low timeout for quick, yet incomplete, results.

6 Mobile Application Performance

We validated the usability of Prometheus as a social data management service by developing a mobile social application, CallCensor, that utilizes the Prometheus inference functions through the exposed API, under real-time constraints. Past work (ContextPhone [39]) described the ContextContact application which offers cues to the caller about the callee's social context such as location, collocation with other people, phone ringer status, etc. Our application builds on these lessons and allows the callee's phone to adjust the phone ring based on the owner's social context and the social relationship with the caller. We measured its end-to-end performance using a real multi-graph of 100 users.

The CallCensor application leverages social information received from Prometheus to decide whether or not to allow incoming calls to go through. For each incoming call, the application queries Prometheus with a *social_strength* or *neighborhood* inference request to assess the type of social connection between the caller and the phone owner. Based on the owner settings (e.g., don't allow personal calls while at work), the application decides if the phone should ring, vibrate or silence upon receiving the call. The application was written in Java for devices running Google Android OS and was tested on a Nexus One mobile phone from HTC (1GHz CPU, 512MB RAM).

There are multiple scenarios a caller can be connected to the phone's owner; we tested three: directly connected within 1 social hop, indirectly connect by 2 social hops, and connected with a high social strength. We tested each of these scenarios 50 times. For each of them, the *ego* and *alter* were randomly chosen, and the inference request was sent to a random peer. We assumed users defined ACPs allowing access to all their

data, thus enabling requests to proceed over multiple hops, and consequently stressing the system at maximum possible load. We measured the end-to-end response time of an inference request submitted to Prometheus. This experiment introduced additional overhead due to the communication between the mobile application and Prometheus, and the processing time by the mobile application.

6.1 Social Multi-Graph from Real Traces

The social graph used in the CallCensor application experiments was based on data collected at NJIT [40]. The graph has two types of edges, representing Facebook friends and Bluetooth collocation. Mobile phones were distributed to students and collocation data (determined via Bluetooth addresses discovered periodically by each mobile device) were sent to a server. The same set of subjects installed a Facebook application to provide their friend lists and participate in a survey. The user set was small (100 users) compared to the size of the student body ($9,000$), therefore resulting in a somewhat sparse graph. The collocation data have two thresholds of 45 and 90 minutes for users to have spent together.

While the graph edges were not initially weighted, we applied synthetic weights of 0.1 for "Facebook" edges, 0.1 for "collocation" of 45 minutes and 0.2 for "collocation" of 90 minutes. For the mobile application experiments, we consider the "collocation" edges to represent a work relationship, while the "Facebook" edges represent a personal relationship. The user (*ego*) was assumed to be in a work environment when another user (*alter*) called. As shown in Figure 6, the multi-graph provided for better connectivity between users since neither the "Facebook" nor the "collocation" graph is connected, but the graph containing both types of edges is. We equally distributed the $NJIT$ graph with a social mapping on three PL nodes. Splitting this 100 user-graph on 3 PL peers implies a similar ratio of users/peer as before, while testing 1 and 2 hop inference execution on a real multi-graph.

6.2 Experimental Results

Figure 7 presents the performance for the requests sent by CallCensor, for each of the three scenarios examined. The results show the time spent by the requests only in Prometheus and the overall time needed by the CallCensor to request and handle a response. We first observe that the results meet the real-time constraint of the application: the response must arrive before the call is forwarded to the voicemail of the callee (we used the default voicemail time setting). Second, we notice that the application itself introduced a significant overhead: for example, as much as 100% in the 1–hop *neighborhood* and 50% in the 2–hop *neighborhood* and *social_strength*, due to both communication overhead and execution time on the mobile phone. Third, we confirm the similarity of the *social_strength* results with the *neighborhood* for 2 social hops, as found in the first set of experiments. Forth, the 2-hop request results using this small social graph are similar in performance with the larger 1000-user graph used before (social mapping, 30 users/peer). In particular, more than 80% of requests finished within 5 seconds in both setups, which leads us to believe

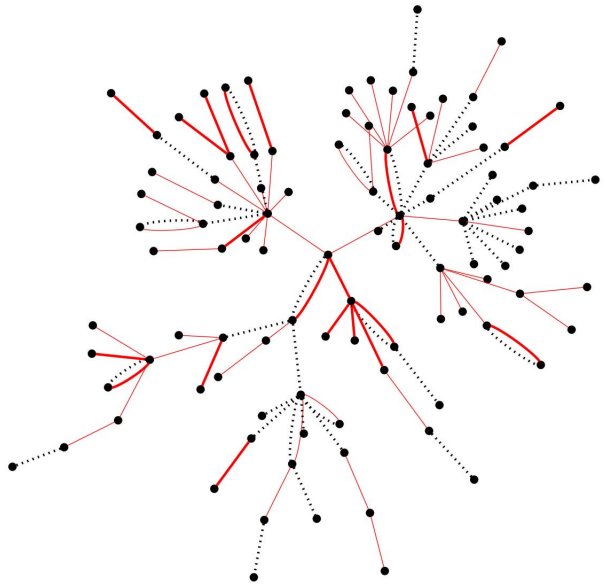


Fig. 6. Real multi-graph with Facebook edges (black dashed lines) and collocation edges (red continuous lines). Line thickness demonstrates edge weight.

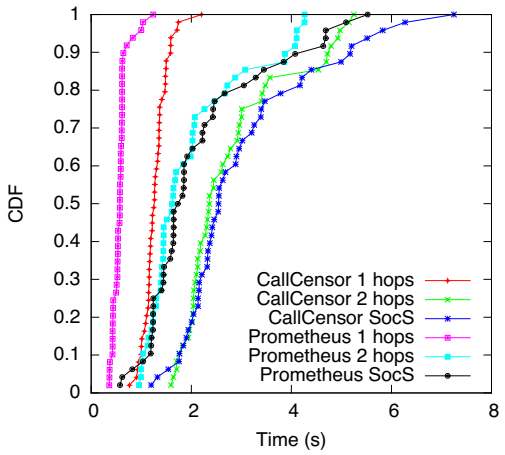


Fig. 7. CDF of average end-to-end response time for CallCensor, under three social inference function requests: 1, 2 hops *neighborhood* and *social_strength (SocS)*.

that similar performance of the mobile application could be expected in a larger social graph distributed over hundreds of peers.

7 Resilience to Malicious Attacks

A social data management service can be the target of attacks at different levels of the system: (1) at the infrastructure level, (2) at the social graph level, (3) at the service level, and (4) at the application level. The defenses against such attacks depend on the system design. In this section we discuss how the design characteristics of Prometheus enable the system to resist such attacks, or mitigate their effects on users.

Attacks at the infrastructure level may attempt to install malcode in the system's software or firmware, or to hinder

access to the service with a denial of service attack. Both centralized and decentralized systems are vulnerable to such attacks. A decentralized system, however, can be the target of additional infrastructure attacks, such as to disrupt system communications (routing of messages between peers) or content storage. Solutions to such attacks in DHT-based systems (such as Prometheus) are reviewed in [41].

Attacks on the social graph aim to manipulate the graph structure by modifying edges, creating new edges with malicious users [42]–[44], or bias honest users to reciprocate edges to attackers [45]. A reciprocal friend edge to an attacker, as implemented in most OSNs, grants him partial or complete access to the victim's social data (i.e., online profile and personal information) and can be used for email spam and phishing campaigns (e.g., [46]). Centralized systems have complete knowledge of the social graph and monitor its changes to detect such malicious activity (e.g., Facebook Immune System [47]). However, the effectiveness of their defense systems is currently limited against socialbot attacks [48]. Prometheus' directed, labeled and weighted social graph is more difficult to manipulate in such attacks because, while it is relatively easy to create an edge from the attacker to the victim, creating a useful reciprocal edge with appropriate label and weight is not in the attacker's control. In the future, Prometheus could employ techniques such as SybilLimit [49] to reduce even further the probability to reciprocate edges to sybil attackers.

Attacks at the service level attempt to manipulate inferences on the social graph, such as drop requests or modify results. Such attacks are more effective on decentralized systems than in centrally controlled systems due to the different level of guarantees that distributed, user-contributed peers can provide. We experimentally elaborate more on this type of attacks in the next subsections.

Finally, in the application level, malicious applications attempt to collect and use social data for spamming, stalking, etc. Users in centralized systems have typically limited or no control on the exposure and use of their data to such 3rd party applications and services, as seen by numerous email spam and phishing campaigns [46], cyberstalking cases [50], as well as collection of private information from online social aggregators (e.g., Spokeo [51]). Instead, Prometheus enables users to control access of applications to their social data via fine-grained ACPs. In addition, application designers can make use of the Prometheus API to reduce attacks on their users' data. For example, an email application could filter spam by allowing incoming emails only when the receiver is reciprocating an edge to the sender with appropriate label and weight, and even using knowledge from the receiver's 2-hop neighborhood to inform the filtering decision [52].

Prometheus is useful to user applications only if it can support an unhindered execution of inference functionalities on users' social data. In the next subsections we explore in more detail how the socially-aware design of Prometheus increases the system resilience to attacks at the service level which attempt to manipulate inference requests.

7.1 Attacks at the Service Level

As explained in Section 4, if a peer does not have the social data necessary to fulfill a request locally, it sends secondary requests to the appropriate peers. The number of such secondary peers contacted depends on the request type and number of hops requested (which can translate to multiple P2P network hops) and how the social graph is decentralized on peers. Further, in a system that protects user privacy such as Prometheus, a requestor cannot distinguish how and from which peer any given item entered the result set. Consequently, for $Alice$ to mount a successful attack at the service level, she would have to control intermediary peers, which have the opportunity to drop incoming requests, modify intermediate results, or change the parameters of secondary requests sent to new peers. Moreover, if malicious peers collude (e.g., they are all owned or have been compromised by $Alice$), they can increase the magnitude of the attack at the service level.

We experimentally investigate how the socially-aware design of Prometheus increases its resilience to these attacks by measuring the opportunity of peers to influence (and thus manipulate) results when serving a *neighborhood* inference request. We define a peer's *influence* on requests as the fraction of requests that the peer serviced over the total number of requests issued in the system. A peer's influence on a request increases with the number of hops the graph is traversed, since, probabilistically, there are more chances for the peer to participate in the request's execution.

We do not consider the first hop (i.e., the source peer) of a request as malicious, since if it is, no results can be considered legitimate. Moreover, we assume trusted peers do not act maliciously to their trusted users. If they do, they can be blacklisted when discovered. We consider the worst case scenario in which we do not restrict the social edge label or weight, all edges are reciprocal, and users define ACPs allowing access to all their data, thus enabling requests to traverse the whole graph.

We extended our preliminary work [53] with two sets of experiments to assess the influence of peers in inference execution on real social graphs. During the experiments, an n-hop neighborhood request was performed for each ego (user) in the social graph. This request was submitted to $ego's$ peer, i.e. peer P_0, which could fulfill requests regarding information *only* about users mapped to P_0. For users in subsequent social hops from ego that P_0 did not have social data, P_0 found the peers storing the particular users' data and submitted secondary requests to be fulfilled by those peers. Each time a peer served a secondary request, we increased the peer's *influence*.

7.2 Peer Influence in Independent Peer Attacks

In our first set of experiments we studied the peer influence on five graphs based on real traces from diverse application domains, such as P2P file sharing ($Gnutella$ [54]), email communications of company employees ($Enron$ [55]), trust on consumer reviews ($Epinions$ [55]) and friendships in a news website ($Slashdot$ [55]). For brevity, we present results for three of them; all results can be found in [56].

We considered all networks undirected and unweighted and used only the largest connected component from each graph to ensure reachability between all pairs of users. The time complexity of the community detection algorithm used in Section 5.1.1 is very high and unsuitable for networks larger than a few thousand nodes. Thus, to produce social mappings of users to peers for the real graphs, we identified social communities with the recursive-based Louvain method used in [57], for average community sizes of N=10, 50 and 100, while maintaining the replication factor to one (K=1). Table 2 presents a summary of these networks and the communities found.

TABLE 2
Summary information of the real graphs used.

Network	Users	Edges	Communities (avg size N)		
			N=10	N=50	N=100
Gnutella04	10,876	39,994	1,087	217	108
Gnutella31	62,561	147,878	6,255	1,245	618
Email-Enron	33,696	180,811	3,369	673	336
Epinions	75,877	405,739	7,564	1,484	727
Slashdot	82,168	504,230	8,206	1,606	793

Figures 8 and 9 plot the CDF of the average influence of peers in 2 and 3 hop requests, for 10 and 50 users/peer respectively, for the *Gnutella04*, *Gnutella31* and *Slashdot* graphs and the 1000-user synthetic graph used in Section 5. The results for all values of N and social graphs can be found in [56]. From these results we formulate the following lessons.

Lesson 4: *The social mapping reduces the average opportunity of peers to influence requests by* 20–50% *for* 2*-hop requests and by* 10–40% *for* 3*-hop requests in comparison to the random mapping.* Figures 8 and 9 show that the random mapping leads to an overall increased opportunity for peers to influence requests, in comparison to the social mapping, regardless of the type and size of graph, and number of hops. This is because in a social mapping, socially-close users are mapped to the same peers, increasing the likelihood that more hops are served locally when compared to randomly mapped users on peers. This implies that a social mapping results in fewer secondary requests sent in the system (regardless of the way communities were identified and mapped on peers). These results support our performance results (Section 5) which showed inferences on a socially-mapped graph to execute faster than on a randomly-mapped graph. Furthermore, increasing the community size increases a peer's influence on requests, regardless of the type of mapping (random or social), since more requests are served by that peer for each of its users. However, this influence increase is more prominent in the random than in the social mapping.

Lesson 5: *The peer influence is affected more by the request's number of hops than the topology or the size of the distributed social graph.* Malicious peers are more effective in small networks, since they can serve and thus influence a larger portion of requests. For example, *Gnutella04* exhibited a similar peer influence profile with *Gnutella31* but with higher influence values. Thus, even though they have the same topological characteristics, in the 6 times larger *Gnutella31* peers have less opportunity to serve (portions of) requests. We also note that *Slashdot*, and similarly *Enron* and *Epinions*, have a different network structure than *Gnutella*, and thus demonstrate a different peer influence profile which stretches to even smaller peer influence values. However, in all graphs and all tested values of users/peer, the decisive factor for the peer influence range is the number of hops requests will attempt to traverse the social graph, as seen from the experimental results for 2 and 3-hop requests.

Lesson 6: *The social mapping allows the formation of "hot-spot" peers who serve many requests, thus becoming attractive targets to attacks.* In the social mapping we observe that for all networks, community sizes and n-hop requests, some peers exhibited significantly higher influence than average. These highly influential peers control more requests flowing through the P2P topology than the average peer. The emergence of such peers can be attributed to users of high social degree centrality, who are closely connected with each other and more likely to be mapped together on the same peer [58] in the social than in the random mapping. These peers can be identified based on the users mapped on them [57] and targeted for quarantine in the early stages of a malware outburst. Alternatively they can be used to disseminate more efficiently security software patches to handle a malicious attack.

7.3 Peer Influence in Peer Collusion Attacks

In the second set of experiments, we investigated the peer influence when peers collude with each other, e.g., they are all controlled by $Alice$. We assume that $Alice$ attempts to recruit peers for her botnets in two ways. First, she targets random peers to control different parts of the network; we refer to this as *random collusion*. Second, she targets a cluster of peers that serve a particular portion of the graph, e.g., neighboring groups in a large professional organization; we refer to this as *social collusion*, where users mapped on the attacked peers are connected with each other over social edges.

For these experiments we used the largest network *Slashdot* and the social and random mappings with average community size of 10 users/peer. We seeded the collusion by selecting 1% random peers (i.e., 1% independent attackers). Then, we iterated over these peers to expand their collusion sets depending on the collusion type (social or random), until the overall malicious peers (fraction C of all peers) across all collusion sets amounted to a specific portion of the total network. We varied C in the range of $10\%, 20\%..., 50\%$.

Figure 10 shows the average influence of peers for 10 repetitions of each of these scenarios. We observe that the average influence measured on collusion groups (SC or RC) is always higher than the average influence of their individual peer members when not colluding with each other (NC). From these results we draw several lessons.

Lesson 7: *Social mappings are more resilient to collusion attacks than random mappings.* Collusion of peers increases their individual effectiveness when attacking the system. However, a random distribution of the social graph onto peers (RM) forces requests to access data from more peers than in a social distribution (SM), and thus allows peers to control and influence a higher portion of inference requests, either if they are colluding or not.

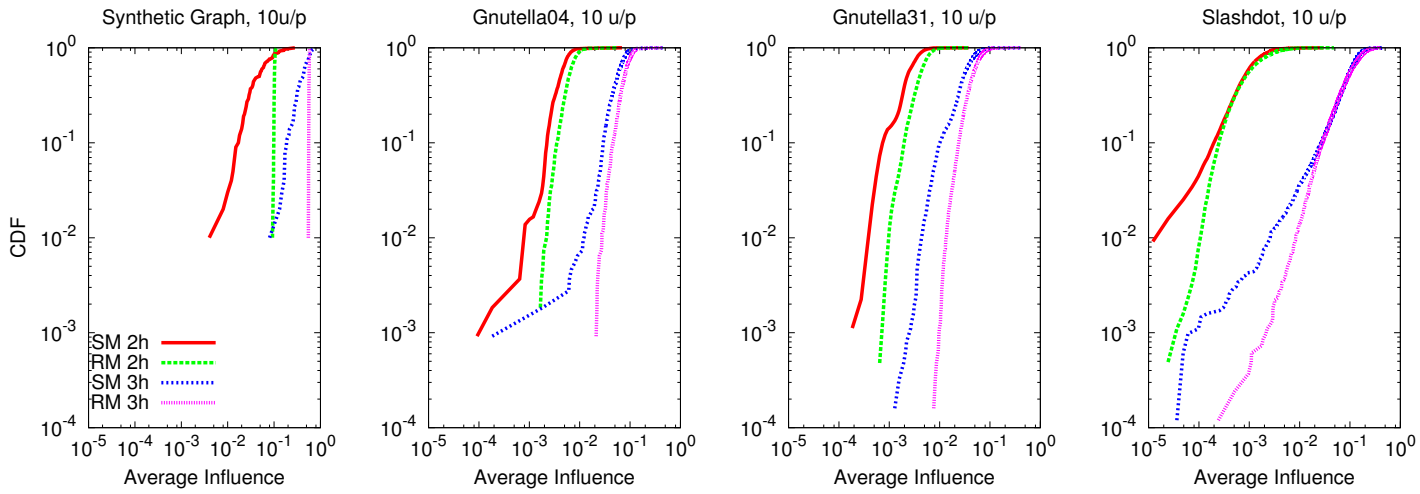


Fig. 8. Peer influence profile: CDF of the average peer influence for the 1000-user synthetic graph, the $Gnutella04$ graph (10.9K users), the $Gnutella31$ graph (62.6K users) and the $Slashdot$ graph (82.2K users), for random (RM) and social mapping (SM) of 10 users/peer, for 2 and 3 hop requests. (Note: axes in log scale).

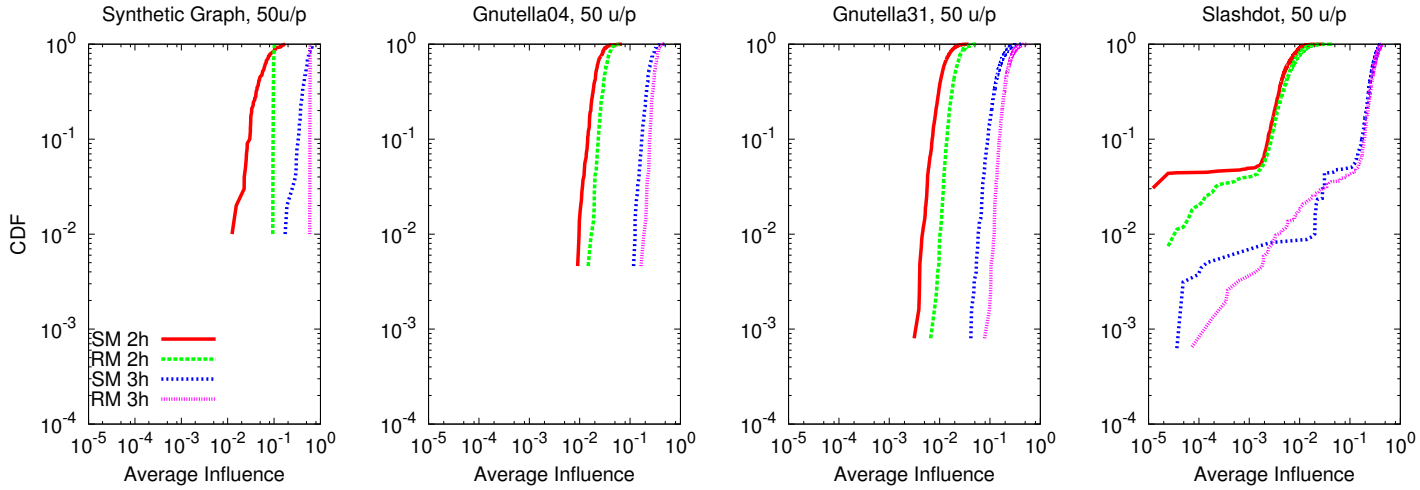


Fig. 9. Peer influence profile: CDF of the average peer influence for the 1000-user synthetic graph, the $Gnutella04$ graph (10.9K users), the $Gnutella31$ graph (62.6K users) and the $Slashdot$ graph (82.2K users), for random (RM) and social mapping (SM) of 50 users/peer, for 2 and 3 hop requests. (Note: axes in log scale).

Lesson 8: *Social peer collusions are less effective on social than random mappings.* In the social collusion, the attack targets neighboring peers (i.e., their users are directly connected in the social graph). In this case, the attacker achieves lower peer influence on requests if the graph was mapped with a social than a random mapping. Overall, the difference between social and random collusion is limited to ~ 0.02 in 2 hops, ~ 0.1 in 3 hops and ~ 0.05 in 4 hops.

Lesson 9: *The peer influence is affected more by the request's number of hops than the collusion size.* In all collusion types and graph mappings, increasing the number of colluding peers increases the average peer influence of an attacker. However, the peer influence depends more on the number of hops the request will traverse the graph, than the collusion size: from less than 0.1 for 2 hops to more than 0.9 for 4 hops (as also shown in the previous experiments). We also observe highest gains on the peer influence rate for a malicious attacker at 3 hop requests than 2 or 4 hops (the 4 hop requests cover most of this graph).

Overall, our experiments demonstrated that the social mapping of user data onto Prometheus peers leads to improved resilience to attacks during request execution, since peers have reduced opportunity to manipulate requests and their results, in comparison to a random mapping (e.g., using a DHT). Furthermore, we observed that a social peer collusion can be more effective than a random peer collusion. Therefore, users should carefully select their trusted peers, since unverified peers from their social neighborhood can potentially have increased influence on their requests than random peers. Finally, there should be default ACPs to control the number of (social and network) hops that requests can travel in the system, since this parameter is more important with respect to peer influence

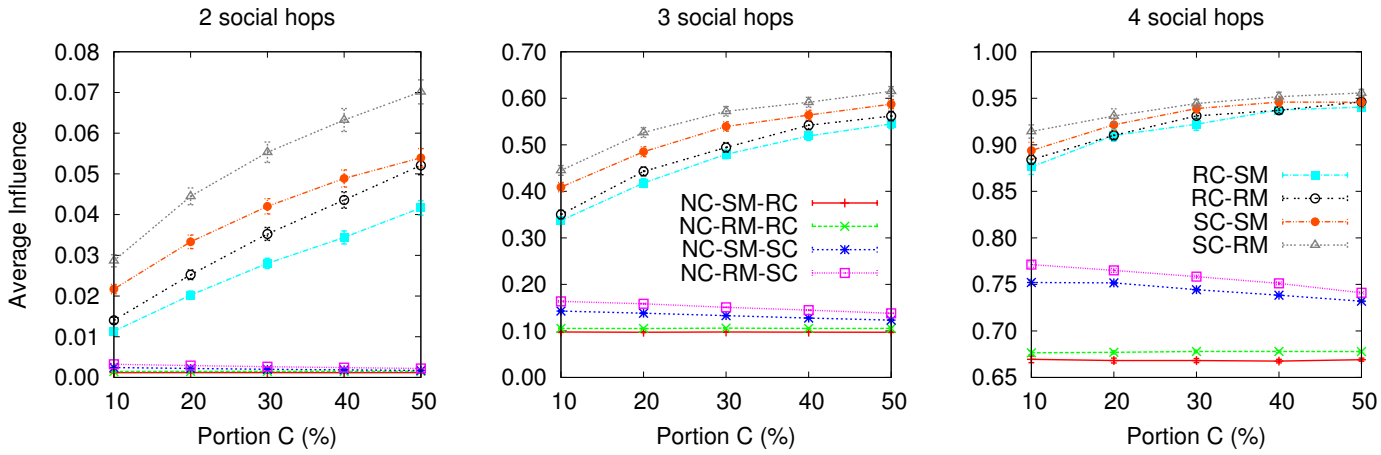


Fig. 10. Average peer influence for random (RM) and social mapping (SM) of the $slashdot$ network, for 10 users/peer and 2, 3 and 4 hop requests. We vary the percentage of $social$ (SC) or $random$ collusion (RC) and compare with the no-collusion (NC) scenario. Y-bars show 95% confidence intervals for 10 repetitions.

and data exposure, than the extend of peer collusion.

8 Conclusions

This paper presented Prometheus, a P2P service that provides decentralized, user-controlled, social data management. Its directed, weighted, labeled and multi-edged social graph offers a fine-grained representation of the users' social state. It enables novel socially-aware applications to mine this rich graph via an API that executes social inferences, while enforcing user-defined access control policies.

We built and evaluated Prometheus using a large distributed testbed and realistic workloads. Prometheus is designed as an application-oriented platform. Thus, we tested its end-to-end performance and showed that deadlines set by application requests can be met without significant reduction in the quality of results. Additionally, we implemented a proof-of-concept mobile social application that utilizes Prometheus functionalities under real-time deadlines. Further, we investigated the resilience of Prometheus to attacks by malicious users and peers, and established that the socially-aware design of Prometheus constitutes a resilient P2P system that can withstand attacks by malicious peers more effectively than solutions that randomly distribute users' data onto peers. We established that the distributed, directed, labeled and weighted social multi-graph maintained by this service, in most cases, can successfully mitigate attacks by malicious users.

Prometheus' performance can certainly be optimized as we have thus far focused on functionality. As mentioned in the evaluation section, we plan to cache and pre-compute results benefiting from the slow changes that occur in social graphs. Since trusted peers periodically check for new records in the social data files of their users, they can update their local copies and rerun the inferences to ensure consistency of these results. Finally, we plan to expand the set of social inferences to utilize the location and collocation of users.

Acknowledgments

This research was supported by the National Science Foundation under Grants No. CNS 0952420, CNS 0831785 and CNS 0831753. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors. The authors would like to acknowledge the contributions of Joshua Finnis and Paul Anderson in the Prometheus code and PlanetLab experiments.

References

[1] J. S. Kong, B. A. Rezaei, N. Sarshar, V. P. Roychowdhury, and P. O. Boykin, "Collaborative spam filtering using e-mail networks," *Computer*, vol. 39, no. 8, pp. 67–73, 2006.

[2] D. N. Tran, F. Chiang, and J. Li, "Friendstore: cooperative online backup using trusted nodes," in *1st Int. Workshop on Social Network Systems*, 2008, pp. 37–42.

[3] J. Li and F. Dabek, "F2F: reliable storage in open networks," in *5th Int. Workshop on Peer-to-Peer Systems*, 2006.

[4] "Graph API," http://developers.facebook.com/docs/api, 2012.

[5] "OpenSocial," http://code.google.com/apis/opensocial/, 2012.

[6] S. A. Golder, D. Wilkinson, and B. A. Huberman, "Rhythms of social interaction: messaging within a massive online network," in *3rd Int. Conf. on Communities and Technologies*, 2007.

[7] S. B. Mokhtar, L. McNamara, and L. Capra, "A Middleware Service for Pervasive Social Networking," in *1st Int. Workshop on Middleware for Pervasive Mobile & Embedded Computing*, 2009.

[8] A.-K. Pietiläinen, E. Oliver, J. LeBrun, G. Varghese, and C. Diot, "MobiClique: Middleware for Mobile Social Networking," in *2nd Workshop on Online Social Networks*, 2009, pp. 49–54.

[9] E. Sarigol, O. Riva, and G. Alonso, "A tuple space for social networking on mobile phones," in *26th Int. Conf. on Data Engineering*, 2010, pp. 988–991.

[10] A. Toninelli, A. Pathak, and V. Issarny, "Yarta: A middleware for managing mobile social ecosystems," in *6th Int. Conf. on Advances in Grid and Pervasive Computing*, 2011, pp. 209–220.

[11] A. Iamnitchi, J. Blackburn, and N. Kourtellis, "The Social Hourglass: an infrastructure for socially-aware applications and services," *IEEE Internet Computing, Special Issue on Social Networking Infrastructures*, vol. 16, no. 3, pp. 13–23, May/June 2012.

[12] R. Xiang, J. Neville, and M. Rogati, "Modeling relationship strength in online social networks," in *19th Int. Conf. on World Wide Web*, 2010, pp. 981–990.

[13] N. Eagle and A. S. Pentland, "Reality mining: sensing complex social systems," *Personal and Ubiquitous Computing*, vol. 10, no. 4, pp. 255–268, 2006.

[14] B. Wellman, "Structural analysis: From method and metaphor to theory and substance," *Social structures: A network approach.*, pp. 19–61, 1988.

[15] N. Kourtellis, J. Finnis, P. Anderson, J. Blackburn, C. Borcea, and A. Iamnitchi, "Prometheus: User-controlled P2P social data management for socially-aware applications," in *11th Int. Middleware Conference*, Bangalore, India, 2010.

[16] S. Buchegger, D. Schiöberg, L. Vu, and A. Datta, "PeerSoN: P2P social networking: early experiences and insights," in *2nd ACM Workshop on Social Network Systems*, 2009, pp. 46–52.

[17] A. Shakimov, A. Varshavsky, L. Cox, and R. Cáceres, "Privacy, cost, and availability tradeoffs in decentralized OSNs," in *2nd Workshop on Online Social Networks*, 2009, pp. 13–18.

[18] L. Cutillo, R. Molva, and T. Strufe, "On the security and feasibility of Safebook: A distributed privacy-preserving online social network," *Privacy and Identity Management for Life*, vol. 320, pp. 86–101, 2010.

[19] L. M. Aiello and G. Ruffo, "LotusNet: Tunable privacy for distributed online social network services," *Computer Communications*, vol. 35, no. 1, pp. 75–88, December 2012.

[20] K. Graffi, C. Gross, D. Stingl, D. Hartung, A. Kovacevic, and R. Steinmetz, "LifeSocial.KOM: A secure and P2P-based solution for online social networks," in *Consumer Communications and Networking Conference*, 2011.

[21] "Enthinnai," http://www.enthinnai.com/, 2012.

[22] "Freedombox," http://freedomboxfoundation.org/, 2012.

[23] "Diaspora," https://joindiaspora.com/, 2012.

[24] A. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility," in *18th Symp. on Operating Systems Principles*, 2001, pp. 188–201.

[25] ——, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *1st Int. Conf. on Distributed Systems Platforms*, 2001, pp. 329–350.

[26] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron, "Scribe: A large-scale and decentralized application-level multicast infrastructure," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 8, pp. 1489–1499, 2002.

[27] P. Anderson, N. Kourtellis, J. Finnis, and A. Iamnitchi, "On managing social data for enabling socially-aware applications and services," in *3rd ACM Workshop on Social Network Systems*, 2010.

[28] J. Golbeck, "The dynamics of web-based social networks: Membership, relationships, and change," *First Monday*, vol. 12, no. 11, Nov 2007.

[29] S. G. B. Roberts and R. I. M. Dunbar, "The costs of family and friends: an 18-month longitudinal study of relationship maintenance and decay," *Evolution and Human Behavior*, vol. 32, no. 3, pp. 186–197, 2010.

[30] N. E. Friedkin, "Horizons of observability and limits of informal control in organizations," *Social Forces*, vol. 62, no. 1, pp. 57–77, 1983.

[31] "Freepastry," http://www.freepastry.org/, 2012.

[32] A. Sala, L. Cao, C. Wilson, R. Zablit, H. Zheng, and B. Y. Zhao, "Measurement-calibrated graph models for social network experiments," in *19th Int. Conf. on the World Wide Web*, 2010, pp. 861–870.

[33] C. Wilson, B. Boe, A. Sala, K. P. N. Puttaswamy, and B. Y. Zhao, "User interactions in social networks and their implications," in *4th Eur. Conf. on Computer Systems*, 2009, pp. 205–218.

[34] P. Anderson, "GeoS: A service for the management of geo-social information in a distributed system," Master's thesis, University of South Florida, 2010.

[35] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *National Academy of Sciences of USA*, vol. 99, no. 12, pp. 7821–7826, 2002.

[36] B. Krishnamurthy, P. Gill, and M. Arlitt, "A Few Chirps About Twitter," in *1st Workshop on Online Social Networks*, 2008, pp. 19–24.

[37] L. Guo, S. Chen, Z. Xiao, E. Tan, X. Ding, and X. Zhang, "Measurements, analysis, and modeling of BitTorrent-like systems," in *5th Conf. on Internet Measurement*, 2005, pp. 35–48.

[38] Z. King, J. Blackburn, and A. Iamnitchi, "BatTorrent: A Battery-Aware BitTorrent for Mobile Devices," in *11th Int. Conf. on Ubiquitous Computing, Poster Session*, 2009.

[39] M. Raento, A. Oulasvirta, R. Petit, and H. Toivonen, "ContextPhone: A prototyping platform for context-aware mobile applications," *Pervasive Computing, IEEE*, vol. 4, pp. 51–59, 2005.

[40] S. J. Pan, D. J. Boston, and C. Borcea, "Analysis of fusing online and co-presence social networks," in *2nd Workshop on Pervasive Collaboration and Social Networking*, 2011.

[41] G. Urdaneta, G. Pierre, and M. van Steen, "A survey of DHT security techniques," *ACM Computing Surveys*, vol. 43, no. 2, Jan 2011.

[42] J. Baltazar, J. Costoya, and R. Flores, "The real face of KOOBFACE: The largest web 2.0 botnet explained," http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp_the-real-face-of-koobface.pdf, Trend Micro Research, Tech. Rep., July 2009.

[43] www.zdnet.com, "56th variant of the koobface worm detected," http://blogs.zdnet.com/security/?p=3414, 2009.

[44] www.news.cnet.com, "Koobface virus hits facebook," http://news.cnet.com/koobface-virus-hits-facebook/, 2008.

[45] Z. Coburn and G. Marra, "Realboy: believable twitter bots," http://ca.olin.edu/2008/realboy/index.html, 2008.

[46] Symantec, "Social network attacks surge," http://www.symantec.com/connect/blogs/social-network-attacks-surge, 2011.

[47] T. Stein, E. Chen, and K. Mangla, "Facebook immune system," in *4th ACM Workshop on Social Network Systems*, Salzburg, Austria, April 2011.

[48] Y. Boshmaf, I. Muslukhov, K. Beznosov, and M. Ripeanu, "The socialbot network: When bots socialize for fame and money," in *27th Annual Computer Security Applications Conference (ACSAC)*, December 2011.

[49] H. Yu, P. B. Gibbons, M. Kaminsky, and F. Xiao, "Sybillimit: A near-optimal social network defense against sybil attacks," in *IEEE Symposium on Security and Privacy*, 2008, pp. 3–17.

[50] WHO@, "Online Harassment - Cyberstalking Statistics 2000-2011," http://www.haltabuse.org/resources/stats/Cumulative2000-2011.pdf, 2012.

[51] Spokeo, http://www.spokeo.com/, 2012.

[52] S. Garriss, M. Kaminsky, M. J. Freedman, B. Karp, D. Mazières, and H. Yu, "Re: Reliable Email," in *3rd Conf. on Networked Systems Design and Implementation*, 2006, pp. 297–310.

[53] J. Blackburn, N. Kourtellis, and A. Iamnitchi, "Vulnerability in socially-informed peer-to-peer systems," in *4th ACM Workshop on Social Network Systems*, Salzburg, Austria, 2011.

[54] M. Ripeanu, A. Iamnitchi, and I. Foster, "Mapping the Gnutella network," *Internet Computing, IEEE*, vol. 6, no. 1, pp. 50–57, 2002.

[55] J. Leskovec, "Stanford large network dataset collection," 2012. [Online]. Available: http://snap.stanford.edu/data/

[56] N. Kourtellis, "On the Design of Socially-Aware Distributed Systems," Ph.D. dissertation, University of South Florida, 2012.

[57] N. Kourtellis and A. Iamnitchi, "Inferring peer centrality in socially-informed peer-to-peer systems," in *11th IEEE Int. Conf. on Peer-to-Peer Computing*, Kyoto, Japan, 2011.

[58] X. Shi, M. Bonner, L. Adamic, and A. C. Gilbert, "The very small world of the well-connected," in *19th Conf. on Hypertext and Hypermedia*, Pittsburgh, PA, USA, 2008.